MEMORIA PRÁCTICA 4

GR2-6

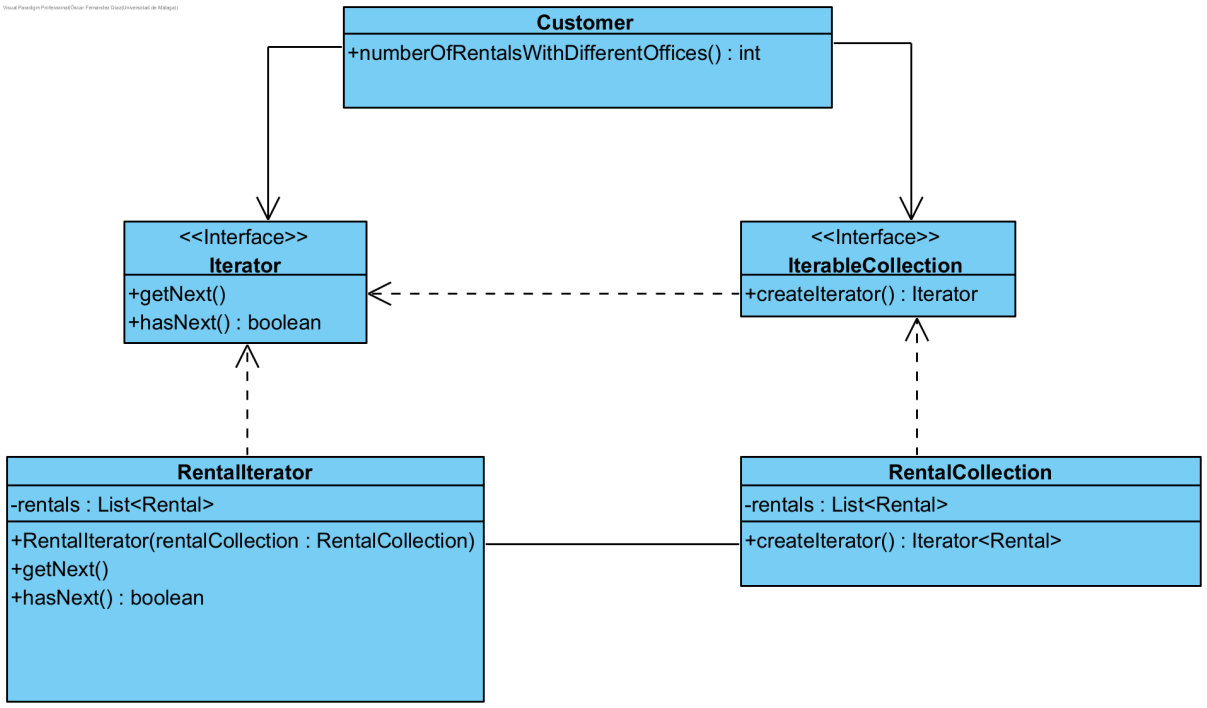
1. Ejercicio 1

Supongamos ahora que queremos definir una operación **numberOfRentalsWithDifferentOffices(): Integer** en la clase Customer que devuelve el número de alquileres web que ha hecho un cliente self donde la oficina de recogida y de entrega es diferente. En este momento del diseño del sistema todavía no sabemos qué estructura de datos utilizaremos para guardar los alquileres que ha hecho/hace un cliente. Se pide:

a) ¿Qué patrón de diseño utilizarías para diseñar esta operación?

Partiendo de que el enunciado no indica qué estructura de datos se usará para almacenar los alquileres de un cliente, creemos que el patrón más adecuado es el ***Patrón Iterador*** ya que el patrón proporciona una forma más eficiente y flexible de recorrer una colección de objetos sin exponer su representación interna, es decir, este patrón nos permite recorrer la colección de alquileres de manera uniforme, independientemente de si se implementa como una lista, conjunto, o cualquier otra estructura.

b) Muestra la parte del diagrama de clases que se ve modificada como resultado de la aplicación del patrón utilizado.



c) Muestra el código Java de la operación *numberOfRentalsWithDifferentOffices():Integer* de la clase *Customer*.

```
public Integer numberOfRentalsWithDifferentOffices() {

   int count = 0;
```

```
    Iterator iterator = rentalCollection.createIterator();


    while (iterator.hasNext()) {
        Rental rental = iterator.getNext();
                    if (rental instanceof WebRental web &&
!web.getDeliveryOffice().equals(web.getPickupOffice())) {
            count++;
        }
    }


    return count;
}
```

El método **numberOfRentalsWithDifferentOffices()** cuenta el número de alquileres realizados por un cliente con diferentes oficinas de recogida y entrega, específicamente en los alquileres web.

La interfaz **Iterator** define los métodos **getNext()** y **hasNext()**, los cuales permiten recorrer los elementos de una colección. La clase **RentalIterator**, que implementa esta interfaz, se encarga de iterar sobre una lista de alquileres (**Rental**), proporcionando la lógica para obtener el siguiente elemento y verificar si hay más elementos en la colección. Por otro lado, la clase **RentalCollection** implementa la interfaz **IterableCollection**, creando un iterador específico (**RentalIterator**) para recorrer sus elementos. En la clase **Customer**, el iterador se utiliza para acceder a la colección de alquileres y realizar operaciones sobre ella, como contar cuántos alquileres tienen oficinas de entrega y recogida diferentes.

Para probar la implementación del patrón de diseño seleccionado, hemos desarrollado el siguiente código:

```
package Principal;


import Ejercicio_1.*;
```

```java
import java.time.LocalDate;

import java.util.ArrayList;

import java.util.List;


public class Principal1 {

   public static void main(String[] args) {

       List<Rental> rentals = new ArrayList<>();


       // Crear oficinas de alquiler

        RentalOffice pickUpOffice1 = new RentalOffice("Calle 1, Madrid", 10);

        RentalOffice deliveryOffice1 = new RentalOffice("Calle 2, Barcelona", 15);


        RentalOffice pickUpOffice2 = new RentalOffice("Calle 3, Valencia", 8);

        RentalOffice deliveryOffice2 = new RentalOffice("Calle 4, Sevilla", 12);


       // Crear modelos y coches

       Model model1 = new Model("Ford Fiesta", 50);

       Model model2 = new Model("Toyota Corolla", 60);


       Car car1 = new Car("1234-ABC", model1, pickUpOffice1);

       Car car2 = new Car("5678-DEF", model2, pickUpOffice2);


       // Crear cliente
```

```
        Customer customer = new Customer("12345678A", "Juan Pérez", rentals);


        // Fechas para los alquileres
        LocalDate startDate1 = LocalDate.of(2024, 12, 1);
        LocalDate endDate1 = LocalDate.of(2024, 12, 5);

        LocalDate startDate2 = LocalDate.of(2024, 12, 10);
        LocalDate endDate2 = LocalDate.of(2024, 12, 15);


        // Crear alquileres web con oficinas diferentes
            WebRental webRental1 = new WebRental(startDate1, endDate1, customer, car1, deliveryOffice1, 3);
        rentals.add(webRental1);
            WebRental webRental2 = new WebRental(startDate2, endDate2, customer, car2, deliveryOffice2, 5);
        rentals.add(webRental2);


        // Llamar a la operación para contar alquileres con oficinas diferentes
                    Integer rentalsWithDifferentOffices = customer.numberOfRentalsWithDifferentOffices();
            System.out.println("Alquileres web con oficinas diferentes: " + rentalsWithDifferentOffices);
    }
}
```

2. Ejercicio 2

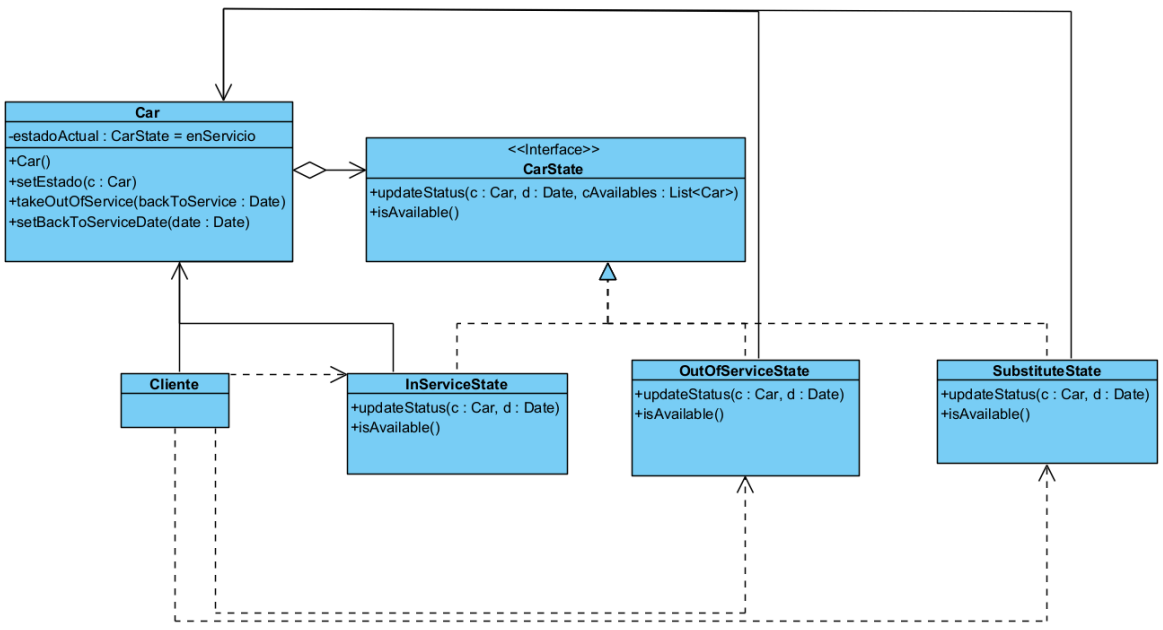
A menudo, los coches que la empresa de alquiler de coches pone a disposición de sus clientes se tienen que poner fuera de servicio (por reparaciones, para pasar la ITV, etc.). Queremos representar esta situación en nuestro sistema para que cuando los coches estén fuera de servicio no puedan ser alquilados aunque registramos, si hay, un coche que lo sustituye. Es por eso que nuestro sistema tendrá que proporcionar dos funcionalidades. La funcionalidad (1) poner un coche fuera de servicio (si ya está fuera de servicio o si está en servicio pero es sustituto de algún coche que está fuera de servicio, la funcionalidad no tendrá ningún efecto). Esta funcionalidad pondrá el coche fuera de servicio y registrará la fecha hasta la cual estará fuera de servicio y, si hay, buscará y registrará también un coche sustituto (será cualquier coche del mismo modelo del coche que se pone fuera de servicio que esté asignado a la misma oficina y que esté en servicio). La funcionalidad (2) pone un coche que estaba fuera de servicio en servicio.

En concreto, nos centraremos en implementar la funcionalidad (1) añadiendo la operación take Out Of Service(backToService:date) de la clase Car. No hace falta implementar la funcionalidad (2).

a) ¿Qué patrón de diseño recomendarías para representar la información descrita (si es que recomiendas alguno)?

   El **Patrón de Estado** se utiliza cuando un objeto necesita cambiar su comportamiento en función de su estado interno, permitiendo que los cambios ocurran de forma dinámica y a lo largo del tiempo. En este caso, el patrón resulta adecuado porque modela como un coche debe cambiar su comportamiento en función del estado en el que se encuentre: **en servicio, fuera de servicio** o **sustituto.**

b) Muestra el diagrama de clases resultante de añadir estas funcionalidades.



c) Muestra el código Java de la operación takeOutOfService de la clase Car.

```java
public void takeOutOfService(Date backToService) {
    if (backToService == null || backToService.before(new Date(System.currentTimeMillis()))) {
        throw new IllegalArgumentException("La fecha no es válida");
    }
    if (this.state.isAvailableToRent()) {
        this.state.updateStatus(this, backToService);
    }
}
```

El método **TakeOutOfService()** valida que la fecha de regreso al servicio sea válida, verifica si el coche está disponible para alquilar y si es así, actualiza el estado a "fuera de servicio", establece la fecha en la que estará disponible de nuevo y por último busca un coche sustituto para este, que será uno del mismo modelo y perteneciente a la misma oficina.

Para implementar esta funcionalidad, hemos diseñado una interfaz llamada **CarState**, que define el comportamiento común a todos los estados posibles de un coche. Esta interfaz incluye dos métodos principales:

1. **isAvailable**: verifica si un coche está disponible para ser alquilado en el estado actual.
2. **updateStatus**: gestiona las actualizaciones del estado del coche.

A partir de esta interfaz hemos creado tres clases que representan los diferentes estados que un coche puede tener:

1. **InServiceState**: Representa el estado en el que el coche está disponible para ser alquilado.
    a. El método **isAvailable** devuelve true.
    b. El método updateStatus cambia el estado del coche a OutOfServieState, establece la fecha en la que volverá a estar disponible y busca un coche sustituto dentro de la lista de coches disponibles que sea del mismo modelo y pertenezca a la misma oficina.

```java
public class InService implements CarState {
    @Override
    public void updateStatus(Car car, Date backToServiceDate) {
        car.setState(new OutOfServiceState());
        car.setBackToServiceDate(backToServiceDate);

        for (Car carSustituto : car.getRentalOffice().getCarsList()) {
            if (carSustituto.getModel().equals(car.getModel())) {
                if (carSustituto.getState().isAvailableToRent()) {
                    carSustituto.setState(new SubstituteState());
                    car.setCarSustituto(carSustituto);
                    return;
                }
            }
        }
    }

    @Override
    public boolean isAvailableToRent() {
        return true;
    }
```

2. OutOfServiceState: Representa el estado en el que el coche no está disponible.
    a. El método isAvailable devuelve false.
    b. El método updateStatus no tiene funcionalidad implementada.

```java
public class OutOfServiceState implements CarState {
    @Override
    public boolean isAvailableToRent() {
        return false;
    }

    @Override
    public void updateStatus(Car car, Date backToServiceDate) {
        //
    }
```

3. SubstituteState: Representa un coche que está temporalmente asignado como sustituto de otro que está fuera de servicio.
    a. El método isAvailable devuelve false.
    b. El método updateStatus no tiene funcionalidad implementada.

```java
public class SubstituteState implements CarState {
    @Override
    public boolean isAvailableToRent() {
        return false;
    }

    @Override
    public void updateStatus(Car car, Date backToServiceDate) {
        //
    }
```

Para la comprobación de la operación hemos creado dos coches con el mismo modelo, hemos quitado el servicio al coche “car1” con matrícula “1234-ABC” y a continuación hemos establecido como coche sustituto el coche “car2” con matrícula “5678-DEF” hemos mostrado por pantalla los datos actualizados y podemos comprobar que la operación se realizó correctamente.

```java
package Principal;


import Ejercicio_2.Car;

import Ejercicio_2.Customer;

import Ejercicio_2.Model;

import Ejercicio_2.RentalOffice;


import java.util.Date;


public class Principal2 {

   public static void main(String[] args) {

          RentalOffice pickUpOffice1 = new RentalOffice("Calle 1, Madrid", 10);


       Model model1 = new Model("Ford Fiesta", 50);

       Car car1 = new Car("1234-ABC", model1, pickUpOffice1);

       Car car2 = new Car("5678-DEF", model1, pickUpOffice1);


              System.out.println("Estado inicial de car1: " + car1.getState().toString());

              System.out.println("Estado inicial de car2: " + car2.getState().toString() + "\n"); // Verifica el estado inicial


             Date date = new Date(System.currentTimeMillis() + 1000000000);

       car1.takeOutOfService(date);


       System.out.println("El coche car1 ha salido de servicio");
```

```
                System.out.println("Estado actual de car1: " +
car1.getState().toString() + "\n");



        car1.setCarSustituto(car2);

           System.out.println("El coche car2 será el sustituo del
coche car1" + "\n");



        car2.takeOutOfService(date);

        System.out.println("El coche car2 ha salido de servicio");

                System.out.println("Estado actual de car2: " +
car2.getState().toString() + "\n");



        if (car1.getCarSustituto() != null) {

                 System.out.println("Coche sustituto de car1: " +
car1.getCarSustituto().getLicensePlate() + "\n");

        } else {

               System.out.println("No hay coche sustituto asignado
para car1.");

        }



        System.out.println("Estado actualizado de car1: " + car1);

         System.out.println("Estado actualizado de car2: " + car2);
// Verifica nuevamente

    }

}
```

Como dato adicional invocamos la función de nuevo pero esta vez para el coche “car2” y como podemos ver por pantalla no realiza ningún efecto ya que sigue en estado “Coche de sustitución”

Salida por pantalla:

```
Estado inicial de car1: En servicio
Estado inicial de car2: En servicio

El coche car1 ha salido de servicio
Estado actual de car1: Fuera de servicio
El coche car2 será el sustituo del coche car1
El coche car2 ha salido de servicio

Estado actual de car2: Coche de sustitución
Coche sustituto de car1: 5678-DEF

Estado actualizado de car1: Matrícula='1234-ABC', Modelo=Ford Fiesta, Estado=Fuera de servicio,
Coche Sustituto=5678-DEF
Estado actualizado de car2: Matrícula='5678-DEF', Modelo=Ford Fiesta, Estado=Coche de sustitución,
Coche Sustituto=No hay coche sustituto
```

3. Ejercicio 3

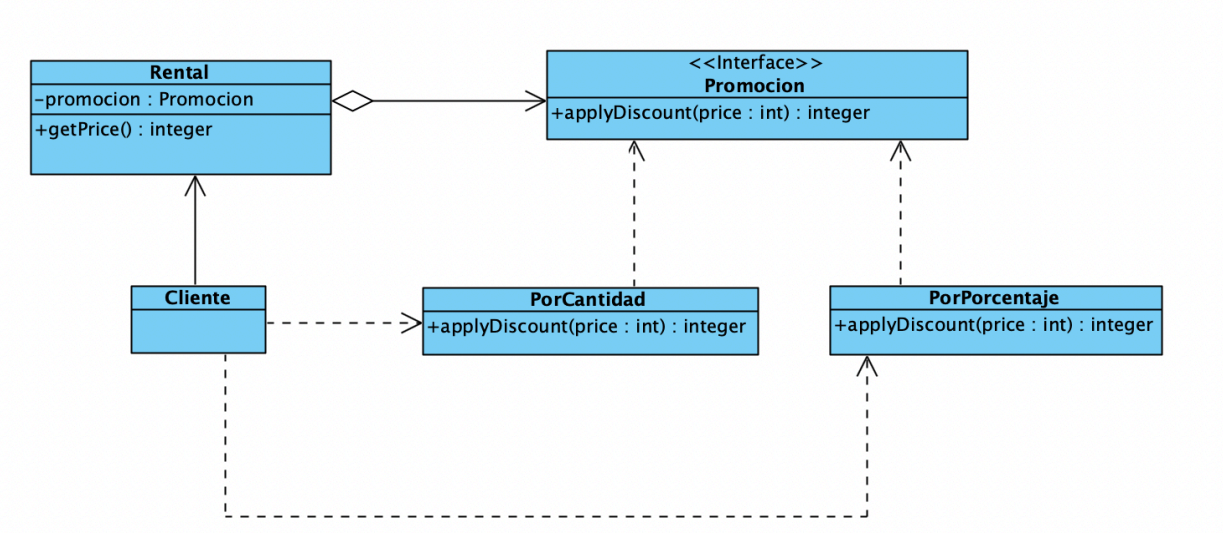
Cuando los clientes de la empresa de alquiler de coches alquilan un coche, el sistema que estamos construyendo tiene que proporcionarles el precio del alquiler. Este precio lo calcula la operación getPrice():Integer de la siguiente forma: El precio base será el precio del modelo del vehículo por día (pricePerDay) * [endDate - startDate] y 2).

Además, la empresa de alquiler de coches puede añadir al cálculo de precios la posibilidad de hacer promociones que implican descuentos de estos precios. Inicialmente, la empresa ofrecerá dos tipos de promociones: por cantidad y por porcentaje. La promoción por cantidad permitirá decrementar el precio del alquiler en la cantidad indicada en la promoción. La promoción por porcentaje decrementará el precio del alquiler en el porcentaje indicado en la promoción. Las promociones se asignan a los alquileres en el momento de su creación. Evidentemente, es posible que a algunos alquileres no se les aplique ninguna promoción. Las promociones que se asignan a los alquileres son determinadas por una política de la empresa que no impacta al diseño de nuestra operación (impactará a la operación que crea los alquileres). Eso sí, la empresa quiere que mientras no se haga el pago del alquiler, si aparecen nuevas promociones, se apliquen a los alquileres siempre y cuando sean más favorables (no nos tenemos que preocupar tampoco de estos cambios, son gestionados por otras operaciones). Se pide:

a) ¿Qué patrón de diseño recomendarías para este caso? Justificar vuestra respuesta.

   El ***Patrón de Estrategia*** permite definir un conjunto de algoritmos o estrategias que pueden intercambiarse dinámicamente en tiempo de ejecución. En este caso, el algoritmo para calcular el precio puede variar dependiendo de la promoción que se aplique. Las promociones (por cantidad y porcentaje) son ejemplos de diferentes estrategias de cálculo.

b) Muestra el diagrama de clases resultante de la aplicación del patrón.



c) Muestra el nuevo código Java de la operación getPrice():Integer

```
public int getPrice() {
    long days = (endDate.getTime() - startDate.getTime()) /
(1000 * 60 * 60 * 24);
    int basePrice = (int)(car.getModel().getPricePerDay() *
days);

    if (promotion != null) {
        return promotion.applyDiscount(basePrice);
    }

    return basePrice;
}
```

El método **`getPrice()`** calcula el precio total de un alquiler basado en el número de días entre dos fechas (**`startDate`** y **`endDate`**) y el precio diario del modelo del coche (**`car.getModel().getPricePerDay()`**). Este método , devuelve directamente el precio base del alquiler sin aplicar descuento en caso de la promoción ser nula. De lo contrario si se aplica un descuento dicho precio base a través de una promoción.

La promoción la hemos implementado a través de la interfaz ***`Promotion`***; ésta define su comportamiento a través del método **`applyDiscount(int price)`**. Para ello, tenemos

dos clases ***PercentagePromotion*** y ***AmountPromotion*** que la implementan y deberán sobreescribir ese método. Estas clases permiten aplicar descuentos basados en porcentaje o cantidad fija, respectivamente.

A través del polimorfismo, el alquiler utiliza la promoción asignada para ajustar el precio según la estrategia implementada.

A continuación, mostraremos la prueba que nos ha permitido realizar la comprobación de dicha operación:

```
public class Principal3 {

   public static void main(String[] args) {

// Crear oficinas de alquiler

         RentalOffice pickUpOffice1 = new RentalOffice("Calle
1, Madrid", 10);

        RentalOffice deliveryOffice1 = new RentalOffice("Calle
2, Barcelona", 15);



         RentalOffice pickUpOffice2 = new RentalOffice("Calle
3, Valencia", 8);

        RentalOffice deliveryOffice2 = new RentalOffice("Calle
4, Sevilla", 12);



        // Crear modelos y coches

        Model model1 = new Model("Ford Fiesta", 50);

        Model model2 = new Model("Toyota Corolla", 60);



        Car car1 = new Car("1234-ABC", model1, pickUpOffice1);

        Car car2 = new Car("5678-DEF", model2, pickUpOffice2);



        // Crear cliente
```

```
        Customer customer = new Customer("12345678A", "Juan
Pérez");



       // Fechas para los alquileres

        Date startDate1 = new Date(2024 - 1900, 11, 1); // 1
de diciembre de 2024

        Date endDate1 = new Date(2024 - 1900, 11, 5);    // 5
de diciembre de 2024



       Date startDate2 = new Date(2024 - 1900, 11, 10); // 10
de diciembre de 2024

       Date endDate2 = new Date(2024 - 1900, 11, 15);    // 15
de diciembre de 2024



       // Crear alquileres web con oficinas diferentes

          WebRental webRental1 = new WebRental(startDate1,
endDate1,    customer,    car1,    deliveryOffice1,    3,    new
AmountPromotion(10));

         WebRental  webRental2  =  new  WebRental(startDate2,
endDate2, customer, car2, deliveryOffice2, 5, null);



       int price1 = webRental1.getPrice();

       int price2 = webRental2.getPrice();

         System.out.println("El precio del alquiler es: " +
price1);

         System.out.println("El precio del alquiler es: " +
price2);

 }

}
```

Por un lado, creamos un objeto ***WebRental1*** (que hereda de ***Rental***) y le asignamos una promoción concreta de cantidad fija 10 (***AmountPromotion***). La función **`getPrice()`**

es invocada sobre el objeto ***webRental1*** y permitirá devolver el precio base del alquiler aplicando un descuento de 10€ al precio total.

Por otro lado, creamos un objeto ***WebRental2*** (que hereda de ***Rental***) y no le asignamos ninguna promoción (**null**). La función **getPrice()** es invocada sobre el objeto ***webRental2*** y permitirá devolver el precio base del alquiler sin aplicar ningún descuento.